CMPSC 465 Data Structures & Algorithms
Fall 2023 Antonio Blanca and Yana Safonova

HW 6

1. (20 pts.) Problem 1

Run DFS on the tree starting from r and store the previsit and postvisit times for each node. Since the given graph is a tree, and we started at the root, the DFS tree is the same as the given tree. We would like to answer queries of the form "is u an ancestor of v?" in $O(1)$ time. For this, note that u is an ancestor of v in the DFS tree if and only if $pre(u) < pre(v) < post(v) < post(u)$. After computing all the pre and post numbers, this condition can be checked for any pair of vertices u and v in $O(1)$ time.

2. (20 pts.) Problem 2

(a) This setting can be represented as a directed graph. We view the intersections as its vertices with the streets being directed edges, since they are one-way. Then the claim is equivalent to saying that this graph is strongly connected. This is true if and only if the graph has only one SCC, which can be checked by running the SCC algorithm in linear time.

(b) The weaker claim says that starting from the town hall, one cannot get to any other SCC in the graph. This is equivalent to saying that the SCC containing the vertex corresponding to the town hall is a sink component. Regarding how the claim can be checked in linear time, there are two possible answers:

(1) First find all the SCCs by SCC algorithm, and then running Explore from the vertex corresponding to the town hall (in the original graph). If any vertices visited through this Explore procedure are not in the SCC that the town hall belongs to, then this SCC is not a sink SCC. Otherwise, it is. The running time is linear because the running time of both the SCC algorithm and Explore is linear, as we know from class.

(2) Run the SCC algorithm. Note that this algorithm for finding SCCs progressively discards sinks SCCs one by one. A component found by the algorithm is a sink if and only if there are no edges going out of the component. We use this to check if the SCC containing the town hall vertex is a sink component. The running time is linear because the running time of the SCC algorithm is linear.

3. (20 pts.) Problem 3

We start by linearizing (topological sorting) G, which we know is possible because it is a DAG. Any path from c to s can only pass through vertices between c and s in the topological order and hence we can ignore the other vertices.

Let $c = v_0, v_1, \ldots, v_k = s$ be the vertices from c to s in the topological order. For each i, we count the number of paths from c to v_i as n_i. Each path to a vertex i and an edge (i, j), gives a path the vertex j and hence

$$n_j = \sum_{(i,j)\in E} n_i$$

Since $i < j$ for all $(i, j) \in E$, this can be computed in increasing order of j. The required answer is n_k. Finding the topological order requires running DFS, which takes $O(|V| + |E|)$ time. Then, computing the sum considers every vertex between c and s in the topological sort, or at most $O(|V|)$ vertices. Specifically, it must retrieve the sum from all of that vertex's already-computed neighbors, so at worst it considers $O(|E|)$ edges total across every sum. Therefore the final running time is $O(|V| + |E|)$.

4. (20 pts.) Problem 4

(a) Assume it is possible to have BFS forward edges. Assume one of this type of edges is (u, w), which leads a node u to its non-child descendant w. As u's descendant, w is visited after u is visited. Due to the edge (u, w), $dist[w] = dist[u] + 1$. So, w is u's child which contradicts with the earlier assumption that w is u's non-child descendant. Therefore, it is impossible to have BFS forward edges.

(b) Algorithm:

- Run BFS on G and obtain the BFS tree. The edges in the BFS tree are tree edges.
- Intuition: both BFS back edges and DFS back edges require the tail to lead to its ancestor as the head. Also, both BFS cross edges and DFS cross edges require the tail to lead to a node that's neither its ancestor nor its descendant as the head. The only difference between BFS edges and DFS edges are that one is regarding the BFS tree and the other is regarding the DFS tree. So, the key to classify BFS back edges and BFS cross edges is to determine the ancestor/descendant relationship between the tail and the head of the edges. We can make use of pre-visit number and post-visit number of each node in a tree to achieve this, and those numbers can be obtained by running DFS on the tree.
- Run DFS on BFS tree to obtain the pre-visit number and post-visit number for each vertex on the BFS tree.
- To classify the edges that are not in the BFS tree (non tree edges), for an edge (w, v), if $pre[v] < pre[w] < post[w] < post[v]$, then it's a back edge. If $pre[v] < post[v] < pre[w] < post[w]$ or $pre[w] < post[w] < pre[v] < post[v]$, then it's a cross edge.

Run time analysis:

- BFS takes $O(|V| + |E|)$ time.
- DFS takes $O(|V| + |E|)$ time.
- Going through the edge set takes $O(|E|)$ time.

Thus, overall, the algorithm takes linear time, $O(|V| + |E|)$.

5. (20 pts.) Problem 5

(a) Notice that in order to have a directed edge from vertex u to vertex v, the clause $\hat{u} \vee v$ needs to be in ϕ. Hence, the edge from u to v is equivalent to $u \Rightarrow v$. Consequently, a path from x to y means that $x \Rightarrow y$. If x and $\hat{x}$ are in the same strongly connected component, then we have $x \Rightarrow \hat{x}$ and $\hat{x} \Rightarrow x$. Then there is no way to assign a value to x to satisfy both these implications.

(b) **Consider the following algorithm:** We recursively find sinks in the graph of strongly connected components and assign true to all the literals in the sink component (this means that if a sink component contains the literal $\hat{x}$, then we assign $\hat{x} = \texttt{true}$ and $x = \texttt{false}$). We will then remove all the variables which have been assigned a value from the graph.

Correctness: The algorithm will output an assignment. Assume on the contrary that the assignment does not satisfy all the clauses, say, a clause $\alpha \vee \beta$ is not satisfied by the assignment, where α and β are literals. So the algorithm sets $\alpha = \texttt{false}$, $\beta = \texttt{false}$, $\hat{\alpha} = \texttt{true}$, $\hat{\beta} = \texttt{true}$. Note that the algorithm sets α and $\hat{\alpha}$ at the same time, and similarly for β and $\hat{\beta}$.

- Case 1 - the algorithm sets α before β: at the time the algorithm sets $\hat{\alpha} = \texttt{true}$, $\hat{\alpha}$ is in a sink component. Since β and $\hat{\beta}$ are set at a later time, the vertices for β and $\hat{\beta}$ must still exist. So, the edge $\hat{\alpha} \Rightarrow \beta$ is still in the graph just before $\hat{\alpha}$ is removed, and hence β must also be in the same sink component. So the algorithm should set $\beta = \texttt{true}$ at the same time it sets $\hat{\alpha} = \texttt{true}$, a contradiction.

- Case 2 - the algorithm sets α and β at the same time: the algorithm sets $\hat{\alpha} = \texttt{true}$ and $\hat{\beta} = \texttt{true}$ at the same time, so they are in the same sink component. Right before they are set, the edge $\hat{\alpha} \Rightarrow \beta$ is still in the graph, so β is in the same component as $\hat{\alpha}$. So β is in the same sink component as $\hat{\beta}$, implying that they are in the same strongly connected component (at the time they are set, and hence in the original graph), a contradiction.
- Case 3 - the algorithm sets α after β: symmetric to Case 1, a contradiction.

(c) To perform the operations in the previous part, we only need to construct this graph from the formula, find its strongly connected components, and identify the sinks - all of which can be done in linear time.

Rubric:

Problem 1, 20 pts

- 10 pts for a DFS from root r
- 10 pts for u is an ancestor of v if and only if $pre(u) < pre(v) < post(v) < post(u)$

Problem 2, 20 pts

(a) 10 points
4 points: correctly formulate the setting as a directed graph.
3 points: correctly state what the mayor claims in graph theory terminology.
3 points: provide an algorithm that can check the claim and explain why the algorithm takes linear time.

(b) 10 points
5 points: correctly state what the weaker claim says in graph theory terminology.
5 points: provide an algorithm that can check this weaker claim and explain why the algorithm takes linear time.

Problem 3, 20 pts

- 10 pts - topological sort
- 5 pts - only consider vertices between the given c and s
- 5 pts - correct sum of path counts

Problem 4, 20 pts

(a) 6 points: provide an explanation, and it makes sense.

(b) 14 points
10 points: provide an algorithm that can correctly categorize the edges in linear time.
4 points: explain why this algorithm takes linear time.

Problem 5, 20 pts

(a) 8pts
- 2pts for if there is an edge from u to v, there should a clause $\hat{u} \vee v$ in ϕ
- 2pts for the path from x to y means $x \Rightarrow y$
- 4pts for conclusion why x and $\hat{x}$ can not be in the same SCC

Or any reasonable explanation worth 8 pts

(b) 8pts - reasonable explanation of why the given algorithm is correct.

(c) 4pts
- 1pts for construct the graph
- 1pts for find the connected components
- 1pts for identify the sinks
- 1pts make the conclusion of linear time